Copyright

by

Joshua Harold Dooms

2010

The Report Committee for Joshua Harold Dooms
Certifies that this is the approved version of the following report:

SQL Database Design Static Analysis

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor: _____
Dewayne Perry

Co-Supervisor _____
Herb Krasner

SQL Database Design Static Analysis

by

Joshua Harold Dooms, B.A.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2010

Dedication

This project is dedicated to my parents without whom; I would never have been able to get through life much less get a master's degree.

Acknowledgements

I'd like to acknowledge Dr. Herb Krasner who was kind enough to guide me through the process of producing this report and Dr. Dewayne Perry who was patient enough to read and advise on this report.

November 22, 2010

Abstract

SQL Database Design Static Analysis

Joshua Harold Dooms, M.S.E.

The University of Texas at Austin, 2010

Supervisors: Herb Krasner, Dewayne Perry

Static analysis of database design and implementation is not a new idea. Many researchers have covered the topic in detail and defined a number of metrics that are well known within the research community. Unfortunately, unlike the use of metrics in code development, the use of these metrics has not been widely adopted within the development community. It seems that a disjunction exists between the research into database design metrics and the actual use of databases in industry. This paper describes new metrics that can be used in industry to ensure that a database's current implementation supports long term scalability, to support easily developed and maintainable code, or to guide developers towards functions or design elements that can be modified to improve scalability of their data systems. In addition, this paper describes the production of a tool designed to extract these metrics from SQL Server and includes

feedback from professionals regarding the usefulness of the tool and the measures contained within its output.

Keywords: SQL, Metric, Software, RDBMS, ORDBMS

Table of Contents

List of Tables

List of Figures

List of Illustrations

No table of contents entries found.

DATABASE DESIGN PATTERNS AND METRICS

Chapter 1: *Introduction*

Relational databases and database management systems serve an integral role in many modern software systems. Their use can greatly simplify the software effort or cause major issues depending on how the database is designed and then subsequently used. Frequently, developers will design their database without considering long term scalability of the application or systems they support. They often see the database as just a tool to store and retrieve data for an application or group of applications. Over the life of the database, as the amount of data increases, the demands on the database increase, and the expectations of the users for performance remain relatively stable, the database can become a serious point of pain for the development organization.

This section has several purposes. The first is to describe database and data system design patterns that can be used to prevent developer and user pain. Next, is describing the current state of RDBMS related metrics and describe why they are not helpful for avoiding poor database design. Following the description of currently defined metrics, this paper defines new metrics that measure how closely a database implementation reflects helpful design patterns. It concludes by describing a process to gather those metrics and describes a plan to confirm that the use of tools for gathering this information are helpful for improving database design practices in industry. Subsequent sections will describe a tool designed to extract these metrics, feedback from industry professionals regarding the tool's use, and conclusions that can be drawn from the feedback.

Chapter 2: *RDBMS Design Patterns and Anti-Patterns*

In order to describe good database design patterns, it may be easier to begin by describing designs that cause long term database scalability issues or anti-patterns. Each anti-pattern description is addressed initially by describing how it causes problems in the system. Then, for each anti-pattern, an alternative pattern is described to address the functional requirement while avoiding the problems.

THE "ONE SIZE FITS ALL" PATTERN

In this pattern, an organization uses a single database design to address the needs of all of its users. An example may help illustrate this point: the Acme Better Widgets Company's ecommerce site. At Acme, they have a need to record their customer and sales data. They have a website for customers to enter their contact, billing, and shipping information and purchase widgets. So, the Acme development team designs the database to support the purchase and shipment of widgets to their customers. This database is designed in BCNF to reduce duplicate data entry. Figure 1 below shows the database design.

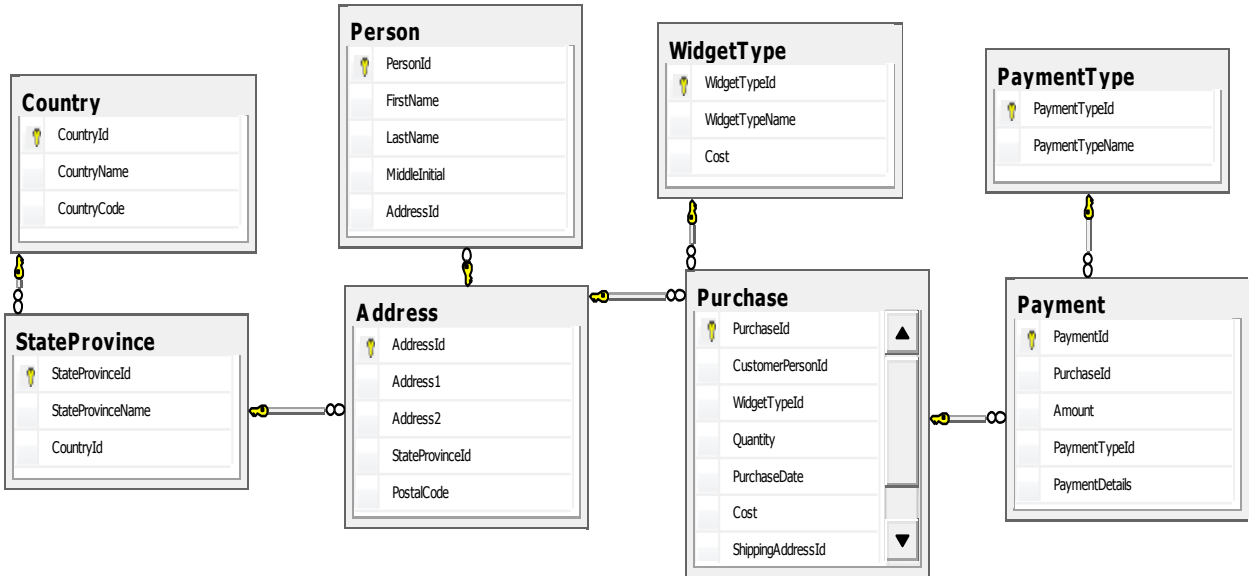

Figure 1: Acme Widgets database design.

For a while, that's all that's needed, but after a few months, upper management realizes that in addition to adding the data to the database for support of online purchases, they also need to analyze the data to help their marketing staff, production staff, and middle management. Acme then produces a web application to support the reporting needs of its internal uses and writes queries that pull from their original database design. At this point, everybody at Acme is happy. The users can enter data and the ecommerce website is responsive while the internal reporting site is also running like a gazelle.

Move forward a couple of years and the number of purchases in the database gets rather large as widgets have been selling like hotcakes. The ecommerce site is still running quickly, but the reports for internal users are taking longer and longer to produce. The development team says "no problem, we'll just add some indices to the tables to support faster querying of the reports" and viola, the reports are now running fast again. Unfortunately, they're now getting reports that the ecommerce site is unresponsive.

Whether or not they realize it, the development team has just encountered one of the biggest issues regarding relational databases. Any design change you make to support fast querying for reporting applications such as more indexing or denormalization, makes data entry and modification slower and vice versa. In order to avoid this pattern, developers need only separate the concerns of data entry (transactional processing) from data retrieval (analytic processing). In order to do this, it could be as simple as creating a copy of the original database when a reporting application is requested and copying the data to the new "reporting" database at predetermined intervals. This allows each of database designs to diverge over time without impacting the performance of the other. As the systems mature, the online transactional processing (OLTP) database will contain more design features that optimize data entry while the online analytical processing (OLAP) database will contain more design features that optimize data retrieval, reporting, and analysis.

THE "PUT THE LOGIC IN THE DATABASE" PATTERN

When designing applications, deployment strategies, and databases, developers will often fall into this trap. Normally, the arguments for designing business logic into the database take one of two forms. The first form is "if we put the logic in a stored procedure (aka sproc) and force the application developers, by reducing database permissions, to use the sproc, we know that the business logic will always be used and the data's integrity will be preserved". The second form is "if we put the logic in the sproc, we can change the behavior of all our applications without having to deploy the software to the servers or clients". By themselves, these are very reasonable claims.

Unfortunately, what they fail to account for is the fact that most relational database systems preserve the properties of atomicity, consistency, isolation, durability (ACID) for transactional processing by serializing incoming requests. What that means is each process making a request to the database may block the requests of any other process until it completes if certain conditions occur. This happens in cases where the reordering of the processes' requests could affect the values persisted in the database or in cases where the values returned to the processes would change depending on ordering. As more processing demands are placed on a database, this effect is increased. In order to avoid this type of blocking, database calls should be kept short and focused on calls that perform only add, update, delete, and select operations. Any other type of logic including flow control, triggers, cursor operations, or dynamic query construction and execution should be placed in a layer of code outside the database to reduce the likelihood that one process will block another.

If developers avoid placing business logic in their database's stored procedures, they can easily add more machines to run the business logic. If these machines aren't concerned with maintaining serial transactional processing and leave that to the database, the cost of scaling the operations will be minimal. Because of this, in a mature database system, the databases' procedures only execute the actions they must to fill their respective roles in the system. Either Create, Read, Update and Delete (CRUD) actions in an OLTP system or Read operations in an OLAP system.

THE "THE LOOSE DESIGN" PATTERN

The last pattern occurs not from any deliberate action, but from a lack of mature processes during the evolution of the database design. Many databases in use today evolved over years and under the hands of many, many developers. Developers can be a crusty lot and can hold onto habits both good and bad like glaciers hold onto water. Because of this, things like naming conventions, the amount of documentation, maintenance of foreign key relationships, and the use of constraints frequently varies greatly depending on the era and/or developer under which the design was produced.

While this won't always directly affect the performance of the database, it almost always affects the code used to consume or add data to the database. An example will help illustrate the point. Imagine that one database developer names their primary keys using the pattern of the table name followed by underscore and the text "KEY" (e.g. WidgetTypes_KEY) while another developer names his keys with the pattern of the table name followed by the text "Id" (e.g. WidgetSoldId) . Notice also that the one database designer pluralized their table name, while the other didn't. While that won't directly affect the performance of the database, any developer writing the code consuming or editing the data in the database has to account for this variation and can't use convention to reduce the complexity of the code. The application developer has to hard code or use data itself to drive the consumption of the data in the database. If convention had been followed relating table name and primary key column name, a simple function like "GetDataByKey(string tableName, object keyValue)" could be used to retrieve any row from any table simply by providing the table name and the key data. Without a consistent convention, the column name(s) defining the key of the table would also have to be passed to the function GetDataByKey.

In addition to the naming conventions, documentation can be added to many database designs directly and can be subsequently read in the database management system (DBMS). This would follow the convention of marrying the implementation and the documentation. When each table has a description of its purpose in the system and each column has a description of its role in the table, subsequent coding of the application becomes much simpler and less error prone. Without this documentation, it's easy for an application developer to misunderstand the role of a table in a system.

For instance, if a table named Person has amongst others, columns named PhoneNumberID and PersonID, there's also table named PersonPhoneNumber with only columns PersonID and PhoneNumberID, and there's also a table named PhoneNumber with columns PhoneNumberID, PhoneNumberTypeID, and Number, it may be hard to figure out what the role is fulfilled by each column in a table. Even given the strict convention followed, while it may seem obvious that the PhoneNumber table holds phone number data, the Person table holds person data, and the two are related to each other through the PersonPhoneNumber table, it's not at all obvious what the PhoneNumberID column in the person table represents. Figure 2 below displays the design described above.

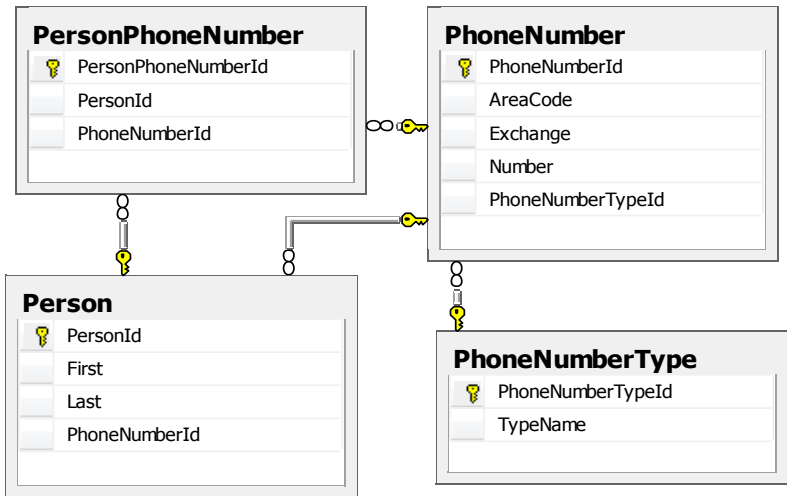

Figure 2: Person phone number design example.

The PhoneNumberID column could be a legacy column before multiple phone numbers were common that hasn't been deleted; the home phone or business phone for the person; the preferred contact phone. Without the documentation, there's really no way to know what the column represents. With a simple description though, it would all be clear and the application developer wouldn't have to scour code to find out what role that column really plays in the system if any.

Consistent use of foreign keys and constraints in the data design greatly reduce the complexity of application code used to produce and consume the data. In general, the more assumptions an application developer can make about the data and its relationships to other data, the simpler the code for consuming the data becomes. One good example is constraining data to be non-null in a column. It may seem like a relatively simple thing, but it's the difference between simply being able to perform operations on a columns data or having to check for null each time data from that column is to be consumed. Another important point to make about the constraints is that they are much easier to apply when a

table or column is first created. If a database designer waits until the design is in use, there may be a considerable cleanup effort necessary before the constraints can be applied.

Chapter 3: *Currently Proposed Metrics*

A number of metrics have been proposed for relational database systems, though the majority seems to focus on object-relational database management systems (ORDBMS). The metrics currently proposed seem to focus mostly on gathering objective data during static analysis of the design without considering on how the data gathered can actually be used to help in the database design or maintenance process. In addition, many of the metrics specific to ORDBMS rather than traditional relational database systems (RDBMS) while much of the industry currently uses RDBMS capabilities with object-relational mapping (ORM) software to serve much of their database need.

ATTRIBUTE AND TYPE METRICS

The metrics themselves can be separated into a hierarchy using the ORDBMS object type to which they relate: attributes, types, columns, tables, or schemas. Among the metrics suggested for attributes are simple attribute size (SAS) which is always one and complex attribute size (CAS) which is the data type size (DTS) of the attribute. For the types, the metrics suggested are the size of the methods in the class (SMC) which is defined in Formalizing Object-Relational Structural Metrics[5] as:

> The SMC function can be obtained in different ways. Possibilities include the use of the fan-in and fan-out metrics proposed by Li and Henry [Li and Henry, 1993], or to consider the method size to be unitary. Our ontology would have to be extended for using the metrics of Li and Henry, since the class Method Specification does not hold all the required information to obtain such metrics.

An additional metric defined for types is the number of hierarchies (NHC) which is either 1 or the number of types that directly inherit from the type. With NHC, SMC, and the sum of the size of the attributes of the class, SAC, defined, the data type size is then defined as the sum of SMC and SAC of the type divided by the NHC of the type. Since DTS is used in defining the complex attribute size, CAS is used in defining the size of the attributes of the class, and SAC is used to define data type size (DTS), data types size is a recursive calculation that requires the size of all types used as attributes in the type be calculated before the type's size can be calculated. That operation recurses until all types resolve to the base types of the system.

COLUMN METRICS

The column oriented metrics are the size of a complex column (CCS) which is defined as the size of a class hierarchy (SHC) divided by the number of columns using the class hierarchy (NCU). "SHC is the Size of the Class Hierarchy (formed by the user-defined data types and their ascendants) upon which the column is defined…"[5] What is meant by this is the sum of the size of every class in the hierarchy from the base class to the class used to define the column.[4] The definition of NCU remains unclear as it may be the number of columns defined upon any type in the entire hierarchy or the number of columns defined with the same data type as the current column.

TABLE METRICS

Unlike the type and column metrics, the table metrics are relatively straightforward. The size of the table (TS) is defined as the sum of the size of the simple columns (TSSC) plus the sum of the size of the complex columns (TSCC). The depth of

the relational tree of the table DRT(T) is defined as the longest relationship tree from table T to any other table in the database.[4] For recursive trees, the depth is only counted until it recurses. The depth of the relational tree of the table should not be confused with the referential degree of the table which is synonymous with the number of foreign keys (NFK) in the table. A few of the table level metrics are self explanatory. Among these are the percentage of complex columns (PCC) and the number of attributes (NA). Unlike these metrics, the number of involved classes (NIC) and the number of shared classes (NSC) require more explanation. NSC(T) is the number of classes used to define columns of table T that are also used to define columns of any other table, while NIC(T) is defined as "This measures the number of all classes that compose the types of the complex columns of T using the generalization and the aggregation relationships."[4].

SCHEMA METRICS

Having defined all of the metrics of the objects that compose the schema, the definition of the schema oriented metrics is relatively straight forward. The size of a system (SS) is the size of all the tables composing the system. The number of attributes (NA) is the sum of all attributes in all tables. The referential degree (RD) is the sum of all foreign keys in all tables. The depth of the referential tree (DRT) is just the longest continuous referential tree defined in the system. The cohesion of the schema (COS) metric is the only exception. COS is defined as the sum of the squares of the number of tables in each unrelated subgraph in the database.[6] How exactly this represents the cohesion of the schema or how this number can be compared across designs is not described in the paper. It would seem difficult to compare across designs given the calculations unbounded nature.

CHALLENGES WITH CURRENTLY PROPOSED METRICS

Many of these metrics were defined as far back as 1999, but are still not commonly used in practice.[6] A number of factors have prevented their widespread adoption over ten years later. The first of challenge with the currently proposed metrics is that many are specific to object-relational database management systems. In practice, databases are best used only for the tasks to which they are best suited: namely, transactional support, data persistence, and data retrieval. By pushing domain and business logic development into layers of software that then interact with the database, practical developers open up new avenues of scalability that would be closed if they used the full object-oriented features of object-relational systems.

In addition, the currently defined metrics, while descriptive, don't seem to have any particular use for which they are well suited. A well designed database is exactly as large and as complex as it needs to be to represent the real world entities to which it refers. Knowing how large or complex the database design is only provides a discrete description of the complexity of the real world referents and the complexity of the real world is likely to be understood before designing the database to represent it. The currently proposed metrics are also not predictive of the time it would take to integrate data from the database into an application since that is often done as piece at a time as needed. The currently proposed metrics also do not provide any insight into how a database design may be improved. In the end, it seems that the currently defined metrics are not used because there is no clear purpose for their existence and the technology that they help describe is unlikely to be widely adopted in the near future.

Chapter 4: *New Metrics Suited to Reinforcing Helpful Design Patterns*

Any new metrics proposed for database design should be defined for currently accepted technology, with specific purpose, and with examples to provide clarity when the meaning is not immediately clear by the definition. In order to begin, it's necessary to define what is meant by currently accepted technology. Over time, this can change, but currently databases are used primarily with native types used for column definitions and methods stored in procedures rather than complex objects. They are currently used to aid in transactional data processing, to preserve integrity of data, to transform data from one form to another, and ultimately to aid in the analysis of the data stored.

The metrics that are proposed here can be separated by the objects to which they pertain: stored procedures, tables, schemas, and systems. While the stored procedures, tables, and schemas are commonly used concepts when talking about relational database management systems, the concept of system is not often discussed or defined. For the purpose of this paper, we can define a system as a group of databases existing on or across servers that provide support to an organization. The data in a system may be related or unrelated and may exist on a single type of database or across multiple types of RDBMS (e.g. sales transactional Oracle database, sales analysis Microsoft SQL database, customer resource management MySQL database). What unifies the databases into a system is the organization that maintains and/or consumes the data is the same organization. By this definition, a single database may be used by multiple organizations and therefore can be a member of multiple systems concurrently.

STORED PROCEDURE METRICS

Length (SPL): the length of a procedure is the number of statements made in the specific RDBMS' query language contained in the procedure. Each SELECT, UPDATE, CREATE, DELETE, INSERT, DROP, DECLARE, and IF statement is included in the count. The prior list is not intended to be exhaustive as much as descriptive of the elements that would be counted. Ultimately, the definition of a statement is left to the RDBMS itself and the values of length will be consistently defined and comparable across designs developed in the same RDBMS. The purpose of this metric is to identify procedures of excessive length. Any procedure of length greater than one could indicate a procedure that could be simplified, but in any given design, the goal would be to focus on the larger most complex procedures first working towards simplifying each until the simplest possible design given the problem space is reached. Given the following pseudo-SQL statement in Figure 3 below, the IF statement counts as one, the DROP counts as well, the CREATE table is the third counted statement, the INSERT INTO followed by the select is the fourth statement, and the final select is the last statement for an SPL of 5. The INSERT INTO followed by the SELECT is not counted as two statements because they are really part of a single insert statement rather than an INSERT and then a SELECT.

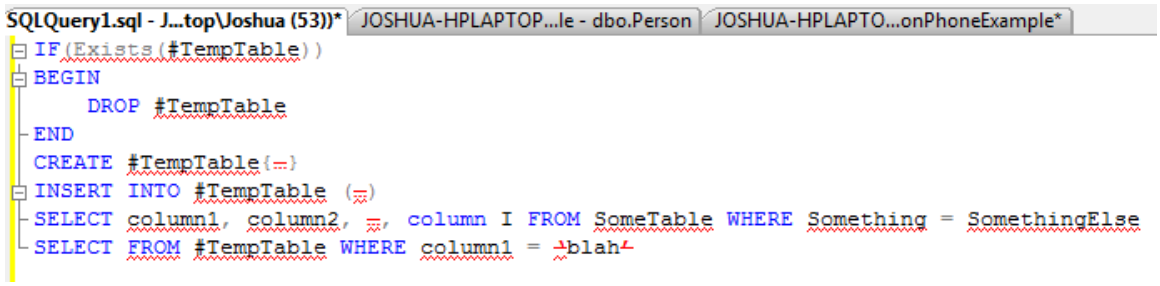

Figure 3: Stored procedure body.

Transactional Orientation (SPTO): transactional orientation is defined as the number of statements, as defined in the RDBMS, that are INSERT, UPDATE, or DELETE divided by the SPL. The purpose of this measure is to help determine where on the continuum between OLTP and OLAP the database falls and will be used when defining schema oriented metrics. Using the statement defined in Figure 3 above, the transactional orientation is 1/5.

Analytic Orientation (SPAO): analytic orientation is defined as the number of statements, as defined in the RDBMS, that are SELECT divided by the SPL. The purpose of this measure is to help determine where on the continuum between OLTP and OLAP the database falls and will be used when defining schema oriented metrics. Using the statement defined above, the analytic orientation is 1/5.

Cyclomatic Complexity (SPCC): cyclomatic complexity was defined by McCabe in 1976 [7] and further refined by Harrison [9]. Because a stored procedure can have multiple exit points, we will use Harrison's definition which is the number of decision points in the program minus the number of exit points plus two. The purpose of defining the complexity of the procedure is to find procedures where flow control is used extensively. Any procedure with a complexity greater than 1 would be a candidate for improvement as the use of flow control statements in procedures is pushes load to the database that would scale more easily if included

in the consuming application code rather than in the database. Given the following stored procedure body below in Figure 4, any RETURN statement as well as the main code body count as an exit point. Since there are no RETURN statements, there is only the one exit point represented by the main code body's exit point. For flow control points in SQL Server, we count any IF, WHILE, or TRY statement as a decision point. TRY blocks are included because the TRY block executes completely or passes control to the CATCH block if a statement produces an error. In the case of the code below, there is one TRY and one IF statement for two total. The complexity of the procedure is therefore 2-1+2=3.

```
SET NOCOUNT ON;
BEGIN TRY
    BEGIN TRANSACTION;
    UPDATE [HumanResources].[Employee]
        SET [Title] = @Title
        ,[HireDate] = @HireDate
        ,[CurrentFlag] = @CurrentFlag
        WHERE [EmployeeID] = @EmployeeID;
    INSERT INTO [HumanResources].[EmployeePayHistory]
        ([EmployeeID]
        ,[RateChangeDate]
        ,[Rate]
        ,[PayFrequency])
        VALUES (@EmployeeID, @RateChangeDate, @Rate, @PayFrequency);
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    IF @@TRANCOUNT > 0
    BEGIN
        ROLLBACK TRANSACTION;
    END
    EXECUTE [dbo].[uspLogError];
END CATCH;
```

Figure 4: Complexity stored procedure body.

TABLE ORIENTED METRICS

Recursive Relationships (TRR): the number of recursive relationships defined on the table. The purpose of this metric is to identify any table with a recursive relationship. Recursive relationships are in themselves complex and in turn, cause complexity in the consuming code. Whenever possible, recursive relationships should be replaced by the use of nested sets. [8]

Percentage of Nullable Columns (TNC): the number of nullable columns in the table divided by the total number of columns in the table. Nullable columns increase the complexity both of the consuming code and the select statements used to access the data. When possible, columns should be made non-null.

Non-commented columns (TNCC): the number of non-commented columns if applicable in the RDBMS. In the case that the RDBMS supports documentation of columns in the database, this is the number of columns that have no documentation. Whenever available, the databases support for integrated documentation should be used.

Not commented (TC): a Boolean indicating whether or not the table is commented if applicable in the RDBMS. In the case that the RDBMS supports documentation of tables in the database, this is whether or not the table has documentation. Whenever available, the databases support for integrated documentation should be used.

Adherence to commenting (TCC): the percentage of columns commented divided by two plus .5 if the table is commented. Essentially, consistently commenting the tables establishes half of the adherence to commenting. Following column commenting conventions establishes the other half of the adherence.

User defined type columns (TUDT): the number of columns using a user defined type as the type of the column. While the RDBMS may support user defined types, use of them can greatly complicate the code used to consume or analyze the data later in the process. Where possible, avoid user defined types.

Percentage of columns without constraints (TCWC): the number of columns with no constraints applied divided by the total number of columns. With few exceptions, there is some expectation about the form of the data in a column and what it means. Whenever possible, constraints should be applied not only to ensure the integrity of the data, but also as a means of documenting the database. This in turn helps to ensure that the code developed to consume the data is as simple as it can be.

Transactional orientation (TTO): the one minus the analytic orientation of the table.

Analytic orientation (TAO): the sum of one plus the number of columns that is a key or datetime then divided by the number of columns or one if all the columns are datetimes or keys. A purely analytic table will be composed of foreign keys or datetime columns and a single data field. A table with that form will have an analytic orientation of one. The further a table diverges from that form, the smaller the analytic orientation value.

Column names violating convention (TVCC): the number of columns that violate the naming convention. This helps identify tables where renaming the tables will help improve the consistency of the design and hence help increase the usability of the design while decreasing the complexity of the consuming code.

Violates naming convention (TVC): a Boolean indicating whether or not the table name follows the established naming convention. Whenever possible, the naming

conventions should be used to increase the design's usability and decrease the complexity of the consuming code.

Adherence to convention (TAC): the percentage of columns following convention divided by two plus .5 if the table name follows convention. Essentially, consistently naming the tables establishes half of the adherence to convention. Following column naming conventions establishes the other half of the adherence.

SCHEMA ORIENTED METRICS

Transactional orientation (STO): twice the mean stored procedure transactional orientation plus the mean table transactional orientation divided by three. This is essentially a weighted average of the transactional orientation of the stored procedures and the tables with twice as much weight given to the stored procedures.

Analytic orientation (SAO): twice the mean stored procedure analytic orientation plus the mean table analytic orientation divided by three. This is essentially a weighted average of the analytic orientation of the stored procedures and the tables with twice as much weight given to the stored procedures.

Number of stored procedures (SNSP): the number of user defined stored procedures in the schema.

Cyclomatic complexity sum (SSCC): the sum of the cyclomatic complexity of all user defined procedures in the schema.

Sum of stored procedure length (SSPL): the sum of the lengths of all user defined procedures in the schema.

Complexity looseness (SCL): one minus the number of stored procedures divided by the cyclomatic complexity sum.

Length looseness (SLL): one minus the number of stored procedures divided by the sum of the stored procedure lengths.

Convention looseness (SACL): one minus the mean of the tables' adherence to convention.

Commenting looseness (SCCL): one minus the mean of the tables' adherence to commenting.

Looseness (SL): complexity looseness plus length looseness plus convention looseness plus commenting looseness divided by four. This value will fall somewhere between 0 and 1. A zero would indicate no looseness and a very clearly and consistently defined schema with short simple procedures. Any value near one would represent little adherence to convention, little commenting, and complex, lengthy procedures.

Maturity (SCM): The sum of one minus the looseness, the absolute value of the transactional orientation minus one half, and the absolute value of the analytic orientation minus one half divided by two. A completely mature schema will have zero looseness and will be either completely transactionally oriented or analytically oriented. In that case, the maturity will be 1. A completely immature schema will have near total looseness and will have neither transactional nor analytic orientation. In that case, the value will be near zero.

SCHEMA ORIENTED METRICS

Maturity (SM): The mean maturity of all schemas in the system.

A TOOL FOR GATHERING DATABASE DESIGN METRICS

Chapter 5: *Introduction*

Microsoft SQL 2008, Oracle, and MySQL database systems all allow for querying a databases' schema information. In order to gather the data, a program could be written that extracts the info required for determining the value of each of the aforementioned metrics from any of those three systems. With implementation for gathering the data from those three systems and presenting the data in a concise report format, the tool could then be distributed to a number of test companies for use. The following describes the implementation of a tool that supports Microsoft SQL Server analysis. Because feedback from professionals regarding the usefulness of the proposed metrics could be strongly influenced by the quality of the tool used to gather those metrics, a description of the tool's implementation is included in addition to the feedback gathered after its use.

Chapter 6: *The Console Tool*

At the heart of the implementation is a Windows console or command line tool. This means that the tool can be run without the use of a user interface. Because development organizations can benefit from integrating the tool as part of their development process, programmatic running and configuration of the tool are important features. If the tool were only able to be run manually through the use of a user interface, it would provide much less value to software development organizations. Figure 5 below shows the tool being run without the use of a GUI.

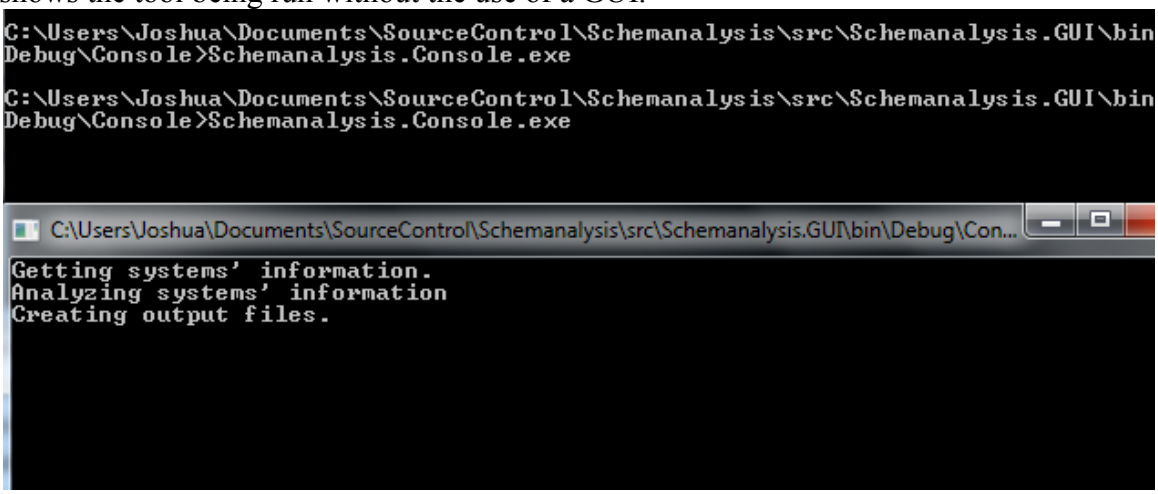

Figure 5: Running the console tool from the command line.

In addition to being executed programmatically, the console tool can be configured programmatically by modifying an xml file that exists in the same directory as the executable. The xml file contains descriptions of data systems including their connection strings, databases, naming conventions and optional filters that allow schemas, tables or procedures to be excluded from the analysis of the system. Also

specified in the xml configuration is the intended location of the output files. Figure 6 below is an example of the xml configuration file.

```
<?xml version="1.0"?>
<configuration>
  <configSections>
    <sectionGroup name="SchemanalysisSection" type="System.Configuration.ConfigurationSectionGroup, System.Configuration, Version=2.0.0.0, Culture=neutral, PublicKey
      <section name="dataSystems" type="Schemanalysis.Configuration.DataSystems, Schemanalysis.Configuration" allowExeDefinition="MachineToApplication"/>
    </sectionGroup>
  </configSections>
  <SchemanalysisSection xmlns="Schemanalysis.Configuration">
    <dataSystems>
      <systems>
        <dataSystem name="Test" tableNamingConvention="^([A-Z]{1}[a-z]+)+$" columnNamingConvention="^([A-Z]{1}[a-z]+)+$"
                    primaryKeyColumnNamingConvention="^([A-Z]{1}[a-z]+)+(ID)$" foreignKeyColumnNamingConvention="^([A-Z]{1}[a-z]+)+(ID)$"
                    tableFilter="" procedureFilter="" schemaFilter="">
          <connectionStrings>
            <connectionString connectionString="Server=JOSHUA-HPLAPTOP\SQLEXPRESS;Database=AdventureWorks2008R2;Trusted_Connection=True;">
              <databases>
                <database name="AdventureWorks"/>
                <database name="AdventureWorks2008R2"/>
                <database name="AdventureWorksDW"/>
                <database name="AdventureWorksDW2008R2"/>
                <database name="AdventureWorksLT"/>
                <database name="AdventureWorksLT2008R2"/>
              </databases>
            </connectionString>
          </connectionStrings>
        </dataSystem>
      </systems>
      <files>
        <applicationFile name="RawDataFile" location="C:\Users\Joshua\Documents\SchemanalysisOutput.xml"/>
        <applicationFile name="FormattedOutputFile" location="C:\Users\Joshua\Documents\SchemanalysisOutput.html"/>
      </files>
    </dataSystems>
  </SchemanalysisSection>
<startup><supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0,Profile=Client"/></startup></configuration>
```

Figure 6: An example of the xml configuration file.

Chapter 7: *The Output Files*

The output of the tool is contained in two files. Both files contain the previously discussed metric data, but one file contains the output in xml form for programmatic consumption while the other contains an html, user-friendly report. One use of the xml data file could be as a trigger for notifications if the databases design falls below certain thresholds determined by the development organization. The html is used primarily as a means to make the use of the program's output more intuitive for new users. The html report is composed of the data in table format with links from the headers to definitions for each metric included at the bottom of the report. In addition to links to the definitions links are used to represent the hierarchical nature of the data. Children of the entity presented in a row are connected to their parent through the use of html page links. In Figure 7 below, any text presented in orange as well as the white text under the cursor represent links to metric definitions or child entity data respectively.

Figure 7: The html output file.

Figure 7 above displays a user clicking on the link for the cyclomatic complexity sum header in a table representing schema data. Clicking on the link will cause the page to navigate the definition of the data contained in the column. Figure 8 below is an example of what the user would see after clicking on the link.

Schema	Cyclomatic Complexity Sum	SSCC	The sum of the cyclomatic complexity of all user defined procedures in the schema.
Schema	Sum of Stored Procedure Length	SSPL	The sum of the lengths of all user defined procedures in the schema.
Schema	Complexity Looseness	SCL	One minus the number of stored procedures divided by the cyclomatic complexity sum.

Figure 8: The metric definitions.

Chapter 8: *The Windows User Interface*

While the tool is completely functional without the user interface, it was included to make the tool's configuration and operation more intuitive and increase the likelihood that developers would adopt it as a part of their development process. The user interface is composed of four controls: the system configuration control, the databases configuration control, the reports control, and the progress control. At any given time, the interface will display the progress control on the bottom pane and one of the other three controls in the top pane. The navigation between the three panes is implemented through links that function to toggle the pane between the three controls.

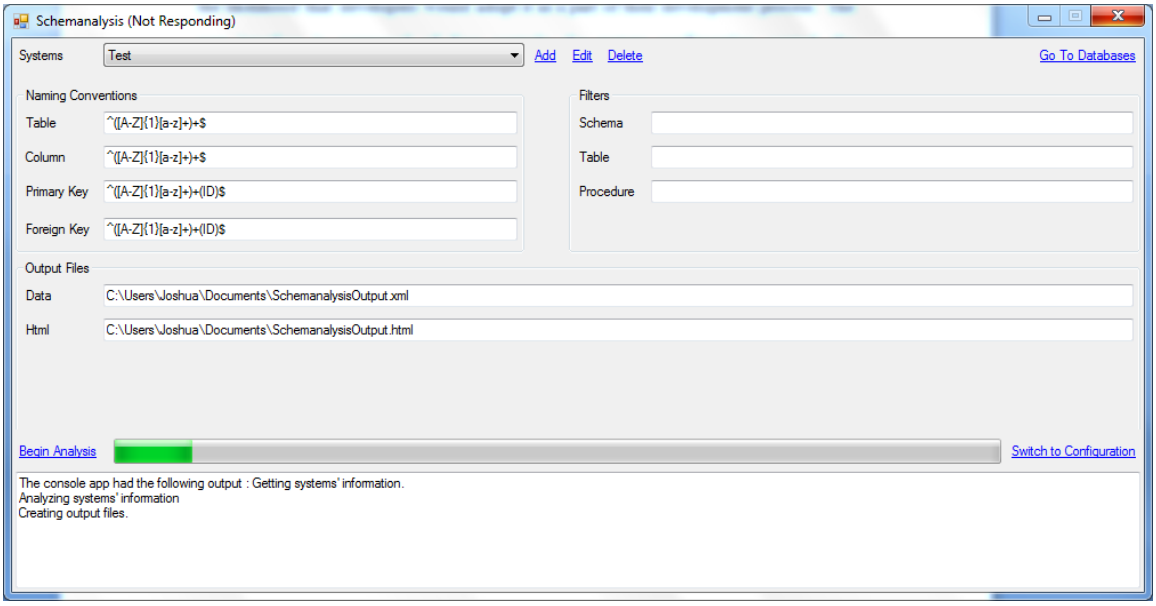

Figure 9: System configuration and progress controls.

Figure 9 above shows the user interface with the system configuration control in the top pane and the progress control in the bottom pane. The system configuration

control allows the user to add, edit or delete systems by name, configure the naming conventions and filters to be used for that system, and edit the location of the output files. The progress control shows the progress of the console tool as a green bar and any output from the console as text below the progress bar. This is the view a user would see after configuring their systems and clicking on the "Begin Analysis" link to the left of the progress bar.

After configuring a system, the user can click on the "Go To Databases" link in the upper right corner of the system configuration control to add and configure connection strings, their child databases, and overrides if needed of the systems level conventions. When finished configuring the databases, the user can either begin analysis since the progress control still populates the lower pane or return to the system configuration by clicking the "Go To Systems" link at the right in Figure 10.

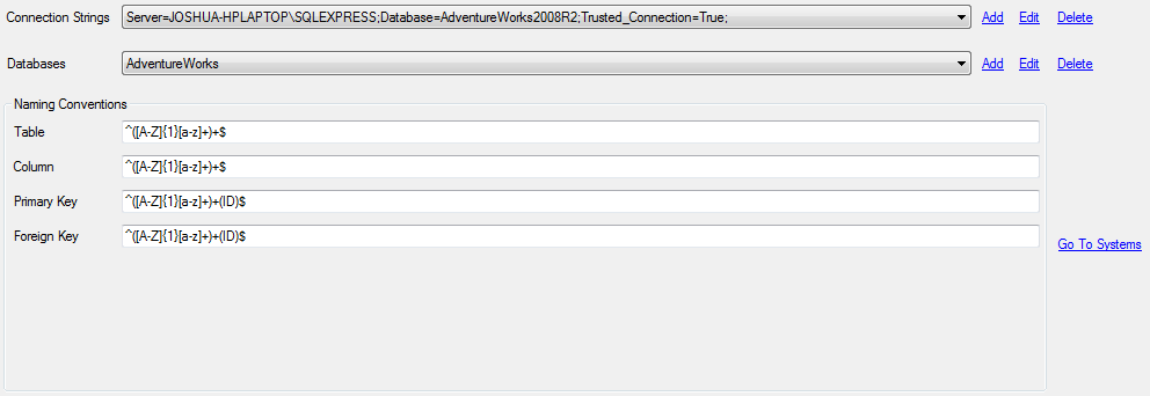

Figure 10: The databases configuration control.

The reports control is simply a web browsing window that displays the html output file in the top pane after the report the analysis is executed. While the html report can be viewed in any web browser, this prevents new users from having to hunt for the

output of their analysis. While the user interface adds no functional value, including it greatly increases the ease with which new users adapt to the use of the tool and may also increase the perceive value of the tool and hence its adoption rate by developers.

FEEDBACK FROM INDUSTRY PROFESSIONALS

Chapter 9: *Introduction*

While it is one thing to propose new metrics regarding database design, it is another to know that the metrics proposed are of real value to professionals currently working in the software development industry. In order to do that, the tool was released to professionals currently working in the industry and their feedback was recorded in a standard format. A summary of that feedback and conclusions based on that feedback are included in this section.

Chapter 10: *The feedback form*

As part of the installation of the previously mentioned tool, a feedback form is added to the same directory as the console tool. The feedback from contains the following questions:

1. Did the software install successfully?
2. Was the user interface intuitive?
3. Of the measures included in the report, on a scale from 1-5 (1 being very helpful and 5 not at all helpful), how helpful are they?
    a. System
        i. Maturity –
    b. Schema
        i. Transactional Orientation (STO)—
        ii. Analytic Orientation (SAO)—
        iii. Number of Stored Procedures (SNSP)—
        iv. Cyclomatic Complexity Sum (SSCC)—
        v. Sum of Stored Procedure Length (SSPL)—
        vi. Complexity Looseness (SCL)—
        vii. Length Looseness (SLL)—
        viii. Convention Looseness (SACL)—
        ix. Commenting Looseness (SCCL)—
        x. Looseness (SL)--
        xi. Maturity (SCM)—
    c. Table

i. Recursive Relationships (TRR)—

ii. Percentage of Nullable Columns (TNC)—

iii. Nullable Foreign Keys (TNFK)—

iv. Non-commented Columns (TNCC)

v. Not Commented (TC)—

vi. Adherence to Commenting (TCC)—

vii. User Defined Type Columns (TUDT)—

viii. Percentage of Columns Without Constraints (TCWC)—

ix. Transactional Orientation (TTO)—

x. Analytic Orientation (TAO)—

xi. Column Names Violating Convention (TVCC)—

xii. Violates Naming Convention (TVC)—

xiii. Adherence to Convention (TAC)--

d. Procedure

i. Length (SPL)--

ii. Cyclomatic Complexity (SPCC)—

iii. Transactional Orientation (SPTO)—

iv. Analytic Orientation (SPAO)—

4. What measures, if any, that were not present in the report, would you like to see included?

5. Could you see using the tool as part of your database development process?

6. What improvements would you suggest to the application?

7. How many systems and/or databases do you have in your organization?

8. Did you uncover any bugs while analyzing your databases?

9. Were any of the measures inaccurate? If so, which one(s)?

10. Did you use the console tool without the user interface?
11. Did you use a programmatic process to build the configuration?
12. Overall, on a scale from 1-5 (1 being very helpful and 5 not at all helpful), how helpful was the tool?
13. If there is any feedback not captured in this document that you'd like to provide, please feel free to do so here.

The questions above were chosen so that the analysis could separate the professional's opinions about the metrics being collected from their opinions regarding the implementation that gathered those metrics. For instance, if the professional couldn't even get the tool to install, that could reduce their perceived value of the metrics presented. In the end, the goal is to evaluate the value of the metrics and not the tool. The tool is simply a means of gathering the data regarding the metrics, so if a correlation exists between the perceived value of the metrics and the perceived value of the tool, the way the professional used the tool, the number of bugs encountered by the professional, etc, the data presented below highlights those correlations.

Chapter 11: *The feedback data*

Level	Metric	Mean	Median	Standard Deviation
System	Maturity	1.4	1	.55
Schema	Maturity	2	2	.71
Schema	Transactional Orientation	1.8	2	.84
Schema	Analytic Orientation	1.8	2	.84
Schema	Number of Stored Procedures	2.4	2	1.67
Schema	Cyclomatic Complexity Sum	3.6	3	.89
Schema	Sum of Stored Procedure Length	2.6	3	1.14
Schema	Complexity Looseness	2.4	2	1.67
Schema	Length Looseness	1.6	1	.90
Schema	Convention Looseness	1.2	1	.45

Level	Metric	Mean	Median	Standard Deviation
Schema	Commenting Looseness	1.2	1	.45
Schema	Looseness	1.8	2	.84
Table	Recursive Relationships	2	2	1.22
Table	Percentage of Nullable Columns	2.2	1	1.79
Table	Nullable Foreign Keys	2	1	1.41
Table	Non-commented Columns	2	2	1.23
Table	Not Commented	2.2	2	1.30
Table	Adherence to Commenting	1.8	2	.83
Table	User Defined Type Columns	2	2	1.23
Table	% of Columns Without Constraints	1.6	1	.89
Table	Transactional Orientation	1.2	1	.45
Table	Analytic Orientation	1.2	1	.45

Level	Metric	Mean	Median	Standard Deviation
Table	Column Names Violating Convention	2	1	1.73
Table	Violates Naming Convention	2	1	1.73
Table	Adherence to Convention	1.2	1	.45
Procedure	Length	2	1	1.42
Procedure	Cyclomatic Complexity	3.4	4	1.52
Procedure	Transactional Orientation	1.8	2	.84
Procedure	Analytic Orientation	1.8	2	.84

Table 1: Statistical data for each metric.

Chapter 12: *Conclusions*

The results of the data only include 5 responders. Most of the metrics proposed in this paper as well as the tool were valued by all of the developers. In addition, every feedback document indicated that the responder would use the tool as part of their development process.

A notable exception to the acceptance of the metrics was the cyclomatic complexity of a stored procedure and the metrics primarily derived from that number. During analysis, it seemed that the cyclomatic complexity varied very little between procedures and most procedures had a complexity of one. When looking at the individual procedures, it became apparent that flow control statements in stored procedures are very often used as a means of directing the flow to specific return statements.

Given that the cyclomatic complexity calculation is the number of decision points minus the number of exit points plus 2, most of the flow control statements were paired with return statements, and there's an implicit default exit point for every stored procedure the value of one returned for most stored procedures makes perfect sense. One of the respondents noted that a different measure for stored procedure complexity was probably necessary and suggested that some combination of number of decision points, the number of parameters, and the stored procedure length would be more indicative of the complexity of stored procedures.

While intuitively, it seems reasonable to treat procedures as code modules with respect to metrics, it seems, based on the feedback, that the cyclomatic complexity metric might not be the best fit for measuring the difficulty of developing and maintaining stored procedures. In order to find the best metric, future research regarding these metrics

would benefit from comparing a number of complexity measures against developer's own assessment of the difficulty of maintaining specific stored procedures and choosing the metric that most closely correlates with those assessments. It's possible that use of an existing metric (e.g. Halstead's Difficulty metric [10]) or a metric developed specifically to address the idiosyncrasies of the SQL language would improve the usefulness of the procedure complexity metric and those metrics including it in their calculation.

An additional commonality between the feedback documents is that every developer thought the descriptions of the metrics while accurate lacked a certain clarification. Each developer suggested that the description of the metrics include some description of what the values for each metric actually meant. They wanted to know what was indicated a "good" or "bad" value in addition to how the metric was actually calculated. In retrospect, this seems like an obvious flaw in the design of the tool and perhaps of this report as a whole.

In order to clarify, the maturity metric varies between zero and one. Values closer to one indicate a well documented system of databases with clear separation of concern between transactional and analytic use of individual databases. In addition, maturity values closer to one indicate that the stored procedures are short, focused methods and the tables and columns follow the specified naming conventions consistently. The looseness metrics also vary between zero and one. Values closer to one indicate better conformance to naming conventions and coding standards. In regard to the transactional and analytic orientation, values also vary between zero and one, but values closer to one or zero indicate a good separation of concerns while values closer to one half indicate less separation between transactional and analytic concerns and are considered poorer design.

Ultimately, the metrics proposed by this report, with the exception of the complexity metric were well received by the developers. Given the tools to easily extract the measures and improvements to the stored procedure complexity metric, it seems quite possible the metrics could become industry standards.

References

1. Codd, E. F. 1974. "Recent Investigations into Relational Data Base Systems," IBM Research Report RJ1385 Republished in Proc. 1974 Congress (Stockholm, Sweden, 1974).
2. Gray, J. September, 1981. "The Transaction Concept: Virtues and Limitations," Proceedings of the 7th International Conference on Very Large Databases. Tandem Computers. pp. 144–154.
3. Baroni, A.L., Calero, C., Piattini, M., and Abreu, F.B. 2005. "A Formal Definition for Object-Relational Database Metrics," In Proceedings of the 7th International Conference on Enterprise Information System.
4. Piattini M., Calero C., Sahraoui H., Lounis H. March 2001. "Object-Relational Database Metrics," L'Object.
5. Baroni, A. L., Calero, C., Ruiz, F., and Abreu, F. B. 2004. "Formalizing Object-Relational Structural Metrics," 5ª Conferência da APSI, Lisbon.
6. Calero, C., Pascual, C., Serrano, and M., Piattini, M. 1999. "Measuring Oracle Database Schemas," IMACS/IEEE CSCC'99 Proceedings, pp. 7101-7107.
7. McCabe, T. J. December, 1976. "A Complexity Measure" IEEE Transactions on Software Engineering, pp. 308-320.
8. Celko, J.J. 2004. "J. Joe Celko's Trees and Hierarchies in SQL for Smarties".
9. Harrison, W.A. October, 1984 "Applying Mccabe's complexity measure to multiple-exit programs," Software: Practice and Experience.
10. Halstead, M.H. 1977. "Elements of Software Science"

Vita

Joshua Harold Dooms received a Bachelor of Arts in Cognitive Science with a focus in Neuroscience from the University of Virginia in 1997. Since 2001, he has worked in the information technology industry as a business analyst, QA analyst, requirements analyst, application developer, and application architect.

Email: joshuadooms@yahoo.com

This report was typed by the author.